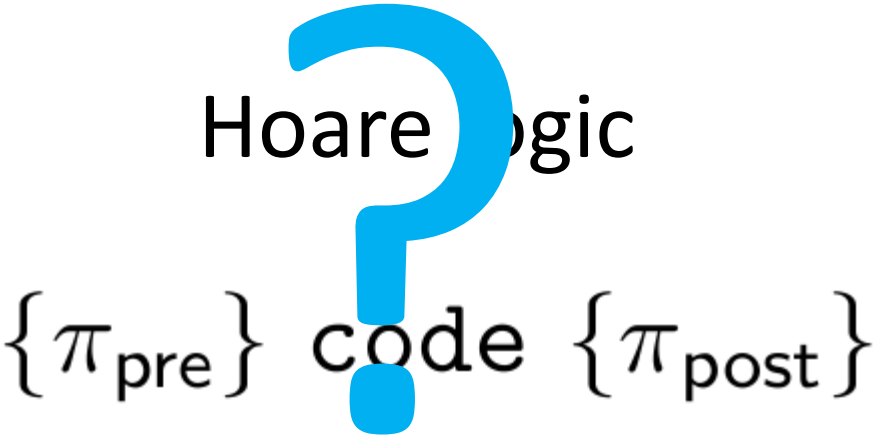
K and Matching Logic

Grigore Rosu
University of Illinois at Urbana-Champaign

Joint work with the FSL group at UIUC (USA)
and the FMSE group at UAIC (Romania)

Question

… could it be that, after 40 years of program verification, we still lack the right semantically grounded program verification foundation?



Current State-of-the-Art in Program Analysis and Verification

Consider some programming language, L

- Formal semantics of L?
  - Typically skipped: considered expensive and useless
- Model checkers for L
  - Based on some adhoc encodings/models of L
- Program verifiers for L
  - Based on some other adhoc encodings/models of L
- Runtime verifiers for L
  - Based on yet another adhoc encodings/models of L
- …

Example of C Program

- What should the following program evaluate to?

```
int main(void) {
  int x = 0;
  return (x = 1) + (x = 2);
}
```

- According to the C “standard”, it is **undefined**
- GCC4, MSVC: it returns **4**
  GCC3, ICC, Clang: it returns **3**
  By April 2011, both Frama-C (with its Jessie verification plugin) and Havoc "prove" it returns **4**

A Formal Semantics Manifesto

- Programming languages must have formal semantics! (period)
  - And analysis/verification tools should build on them at best, or should formally relate to them at worst
    - Otherwise they are adhoc and likely wrong
- Informal manuals are not sufficient
  - Manuals typically have a formal syntax of the language (in an appendix)
  - Why not a formal semantics appendix as well?

Motivation and Goal

- We are facing a semantic chaos
  - Axiomatic, denotational, operational, etc.
- Why so many semantic styles?
  - Since none of them is ideal, they have limitations

- We want a powerful, unified foundation for language design, semantics and verification
  - One semantic approach to serve all the purposes!
  - To work with realistic languages (C, Java, etc.)

Minimal Requirements for an Ideal Language Semantic Framework

- Should be expressive
  - Substitution or environment-based definitions, abrupt control changes (callcc), concurrency, etc.
- Should be executable
  - So we can test it and use it in tools (symb. exec.)
- Should be modular (thus scale)
  - So each feature is defined once and for all
- Should serve as a basis for program reasoning
  - So we can also prove programs correct with it

- Conventional Semantic Approaches -
- Advantages and Limitations –

Chronologically

- 1969: Floyd-Hoare Logic -

- Basis for program verification
- Not executable, and thus, hard to test
  - Semantic errors found by proving wrong properties
  - Soundness rarely or never proved in practice
- Not very expressive
  - Often requires heavy program transformations (e.g., to eliminate side effects, pointers, exceptions, etc.), to reduce languages to cores which can be given an axiomatic semantics
  - Structural program properties (e.g., about heap, stacks, input/output, etc.) hard to state; need special logic support
  - Structural framing (e.g., heap framing) hard to deal with
- Implementations of Floyd-Hoare verifiers for real languages still an art, who few master

- 1971: Denotational Semantics -

Reasonable trade-offs. “Compiles” programs into mathematical objects, so it can be *in principle*:

- Expressive, provided enough/appropriate mathematical domains available
- Executable: we can execute/approximate fixed-points
  - Although factorial(5) crashes Papaspyrou’s C semantics
- Modular, provided one uses advanced features
  - Monads, continuations, resumptions
- Basis for program verification
  - Program = least fixed point, so we can use induction
- Hard to use and understand; requires expert knowledge; no overwhelming evidence it is practical for verification

- 1981: Operational Semantics -

Quite intuitive, easy to understand and define.
Requires minimal training and it scales.

- Executable, by its very nature
  - Although Norish's C semantics not executable, evaluation contexts are inefficient, CHAM has no machine support, etc.
- Expressive … in principle
  - Although hard to use both evaluation contexts (for call/CC, longjumps,…) and environment-store (for pointers, threads,…)
- Modular, when one uses
  - MSOS ideas for dealing with configuration changes, evaluation contexts ideas for control, CHAM ideas for concurrency, etc.
- Considered "too low level", inappropriate for verification

Towards a Better Semantic Approach

- First, we want a semantic framework which, at the same time and uniformly well, is
  - Expressive, Executable, Modular, and
  - Suitable for program reasoning
- Second, we want to develop supporting tools for
  - Defining formal language semantics, and
  - Using the semantics for program verification
    - Put an end to having both operational and axiomatic or other semantics to languages. No more semantics equivalence proofs to be done and maintained!

Starting Point: Rewriting Logic

Meseguer (late 80s, early 90s)

- Expressive
  - Any logic can be represented in RL (it is reflective)
- Executable
  - Quite efficiently; Maude often outperforms SML
- Modular
  - Allows rules to only “match” what they need
- Can serve as a basis for program reasoning
  - Admits initial model semantics, so it is amenable for inductive or fixed-point proofs

Rewriting Logic Semantics Project

- Project started jointly with Meseguer in 2003-4
- Idea: Define the semantics of a programming language as a rewrite theory (set of rules)
- Showed that most executable semantics approaches can be framed as rewrite logic semantics (Modular/SmallStep/BigStep SOS, evaluation contexts, continuation-based, etc.)
  - But they still had their inherent limitations
- Appropriate techniques/methodologies needed

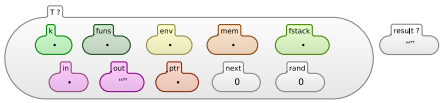
The K Framework
k-framework.org

- A tool-supported rewrite-based framework for defining programming language semantics
- Inspired from rewriting logic
- Used regularly in teaching
- Main ideas:
  - Represent program configurations as a potentially nested structure of cells (like in the CHAM)
  - Flatten syntax into special computational structures (like in refocusing for evaluation contexts)
  - Define the semantics of each language construct by semantic rules (a small number, typically 1 or 2)

Complete K Definition of KernelC

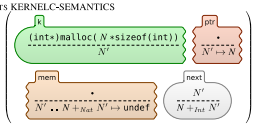
```
MODULE KERNELC-SYNTAX
  IMPORTS K-LATEX+PL-ID+PL-INT
  SYNTAX  Exp ::= Exp + Exp [strict]
                | DeclId
                | Id
                | Int
                | Exp - Exp [strict]
                | Exp ++
                | Exp == Exp [strict]
                | Exp != Exp [strict]
                | Exp <= Exp [strict]
                | Exp < Exp [strict]
                | Exp % Exp [strict]
                | ! Exp
                | Exp && Exp
                | Exp ? Exp : Exp
                | Exp || Exp
                | printf("%d;", Exp ) [strict]
                | scanf("%d",& Exp )
                | scanf("%d", Exp ) [strict]
                | NULL
                | PointerId
                | (int*)malloc( Exp *sizeof(int)) [strict]
                | free( Exp ) [strict]
                | * Exp [strict]
                | Exp [ Exp ]
                | Exp = Exp [strict(2)]
                | Id ( List{Exp} ) [strict(2)]
                | Id ( )
                | random()
                | srandom( Exp ) [strict]
  SYNTAX  Stmt ::= Exp ; [strict]
                | {}
                | { StmtList }
                | if( Exp ) Stmt
                | if( Exp ) Stmt else Stmt [strict(1)]
                | while( Exp ) Stmt
                | return Exp ; [strict]
                | DeclId List{DeclId} { StmtList }
                | #include< StmtList >
  SYNTAX  StmtList ::= StmtList StmtList
                | Stmt
  SYNTAX  Pgm ::= StmtList
  SYNTAX  Id ::= main
  SYNTAX  PointerId ::= * PointerId [ditto]
                | Id
  SYNTAX  DeclId ::= int Exp
                | void PointerId
  SYNTAX  StmtList ::= stdio.h
                | stdlib.h
  SYNTAX  List{Bottom} ::= List{Bottom} , List{Bottom} [assoc hybrid id: () strict]
                | ()
                | Bottom
  SYNTAX  List{PointerId} ::= List{PointerId} , List{PointerId} [ditto]
                | List{Bottom}
                | PointerId
  SYNTAX  List{DeclId} ::= List{DeclId} , List{DeclId} [ditto]
                | DeclId
                | List{Bottom}
  SYNTAX  List{Exp} ::= List{Exp} , List{Exp} [ditto]
                | Exp
                | List{DeclId}
                | List{PointerId}
END MODULE

MODULE KERNELC-DESUGARED-SYNTAX
  IMPORTS K-LATEX
  IMPORTS KERNELC-SYNTAX
  MACRO  ! E = E ? 0 : 1

  MACRO  E1 && E2 = E1 ? E2 : 0

  MACRO  E1 || E2 = E1 ? 1 : E2

  MACRO  if( E ) St = if( E ) St else {}

  MACRO  NULL = 0

  MACRO  I () = I ( () )

  MACRO  int * PointerId = int PointerId

  MACRO  #include< Stmts > = Stmts

  MACRO  E1 [ E2 ] = * E1 + E2

  MACRO  scanf("%d",& * E ) = scanf("%d", E )

  MACRO  int * PointerId = E = int PointerId = E

  MACRO  int X = E ; = int X ; X = E ;

  MACRO  stdio.h = {}

  MACRO  stdlib.h = {}

END MODULE
```

```
MODULE KERNELC-SEMANTICS
  IMPORTS K-SHARED
  IMPORTS K+KERNELC-DESUGARED-SYNTAX+PL-CONVERSION+PL-RANDOM
  CONFIGURATION:
```



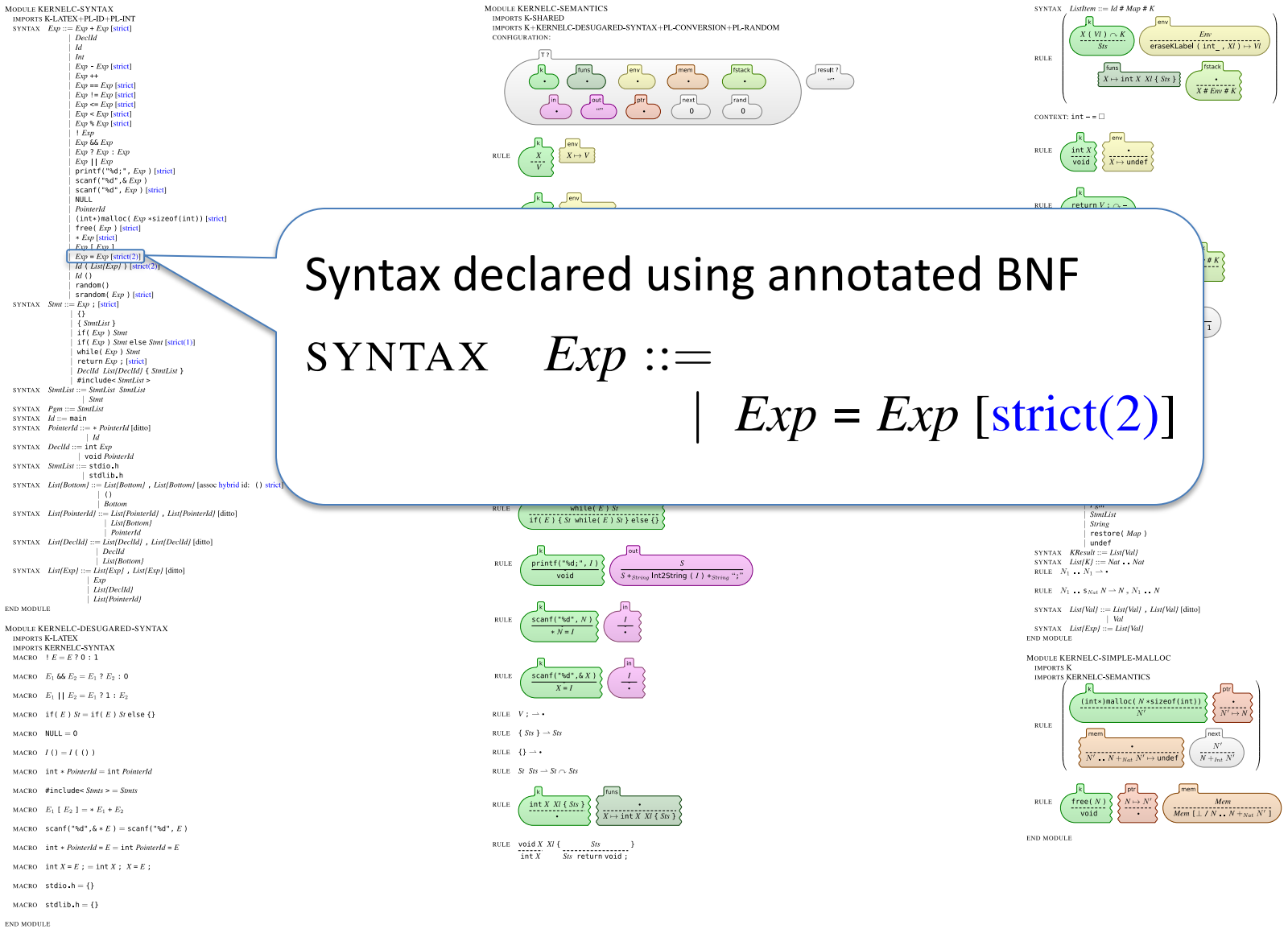

```
  RULE  X  V   [ env: X ↦ V ]

  RULE  X ++  I   [ env: X ↦ I  I +Int 1 ]

  RULE  X = V  V   [ env: X ↦ _  V ]

  RULE  I1 + I2 ⇀ I1 +Int I2

  RULE  I1 - I2 ⇀ I1 −Int I2

  RULE  I1 % I2 ⇀ I1 %Int I2        when I2 !=Int 0

  RULE  I1 <= I2 ⇀ Bool2Int ( I1 ≤Int I2 )

  RULE  I1 < I2 ⇀ Bool2Int ( I1 <Int I2 )

  RULE  I1 == I2 ⇀ Bool2Int ( I1 ==Int I2 )

  RULE  I1 != I2 ⇀ Bool2Int ( I1 !=Int I2 )

  RULE  _?_:_ ⇀ if(_)_else_

  RULE  if( I ) _ else St ⇀ St        when I ==Int 0

  RULE  if( I ) St else _ ⇀ St        when ¬Bool I ==Int 0

  RULE  k: while( E ) St  /  if( E ) { St while( E ) St } else {}

  RULE  k: printf("%d;", I )  /  void        out: S  /  S +String Int2String ( I ) +String ";"

  RULE  k: scanf("%d", N )  /  + N = I        in: I  /  .

  RULE  k: scanf("%d",& X )  /  X = I        in: I  /  .

  RULE  V ; ⇀ •

  RULE  { Sts } ⇀ Sts

  RULE  {} ⇀ •

  RULE  St Sts ⇀ St ↷ Sts

  RULE  k: int X Xl { Sts }  /  •        funs: .  /  X ↦ int X Xl { Sts }

  RULE  void X Xl {  Sts  }  /  int X        Sts return void ;
```

```
  SYNTAX  ListItem ::= Id # Map # K

  RULE  k: X ( Vl ) ↷ K  /  Sts        env: Env  /  eraseKLabel ( int _ , Xl ) ↦ Vl        funs: X ↦ int X Xl { Sts }        fstack: .  /  X # Env # K

  CONTEXT: int _ = □

  RULE  k: int X  /  void        env: .  /  X ↦ undef

  RULE  k: return V ; ↷ _  /  V

  RULE  k: V ↷ .  /  K        env: _  /  Env        fstack: _ # Env # K  /  .

  RULE  k: random()  /  randomRandom ( N' )        rand: N'  /  N' +Nat 1

  RULE  k: srandom( I )  /  void

  CONTEXT: * □ = _

  CONTEXT: * □ ++

  SYNTAX  Val ::= Int
                | void
  SYNTAX  Exp ::= Val
  SYNTAX  K ::= List{DeclId}
                | List{Exp}
                | List{PointerId}
                | Pgm
                | StmtList
                | String
                | restore( Map )
                | undef
  SYNTAX  KResult ::= List{Val}
  SYNTAX  List{K} ::= Nat .. Nat
  RULE  N1 .. N1 ⇀ •

  RULE  N1 .. sNat N ⇀ N , N1 .. N

  SYNTAX  List{Val} ::= List{Val} , List{Val} [ditto]
                | Val
  SYNTAX  List{Exp} ::= List{Val}
END MODULE

MODULE KERNELC-SIMPLE-MALLOC
  IMPORTS K
  IMPORTS KERNELC-SEMANTICS
```

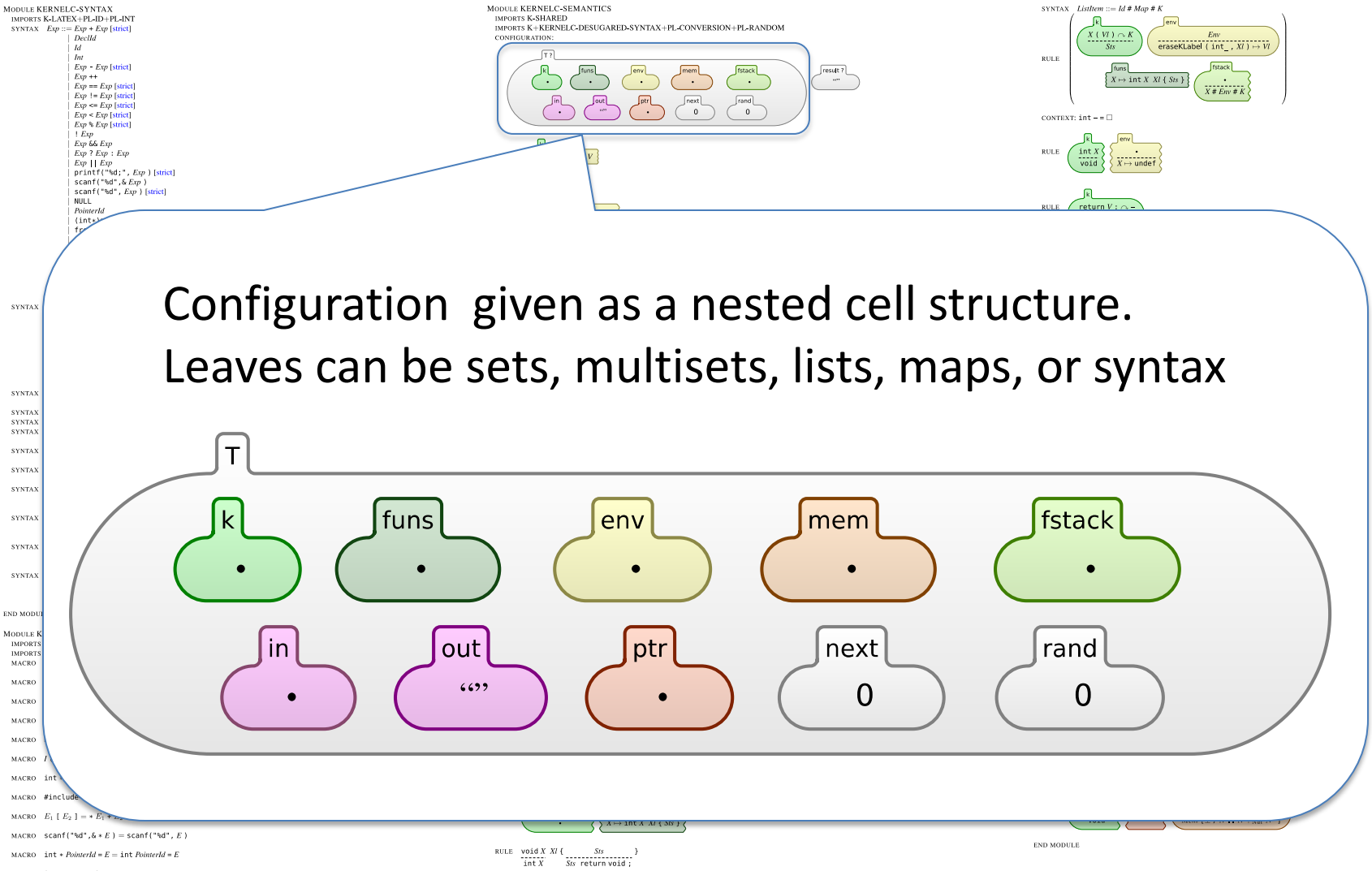



```
  RULE  k: free( N )  /  void        ptr: N ⇀ N'  /  .        mem: Mem  /  Mem [⊥ / N .. N +Nat N' ]

END MODULE
```

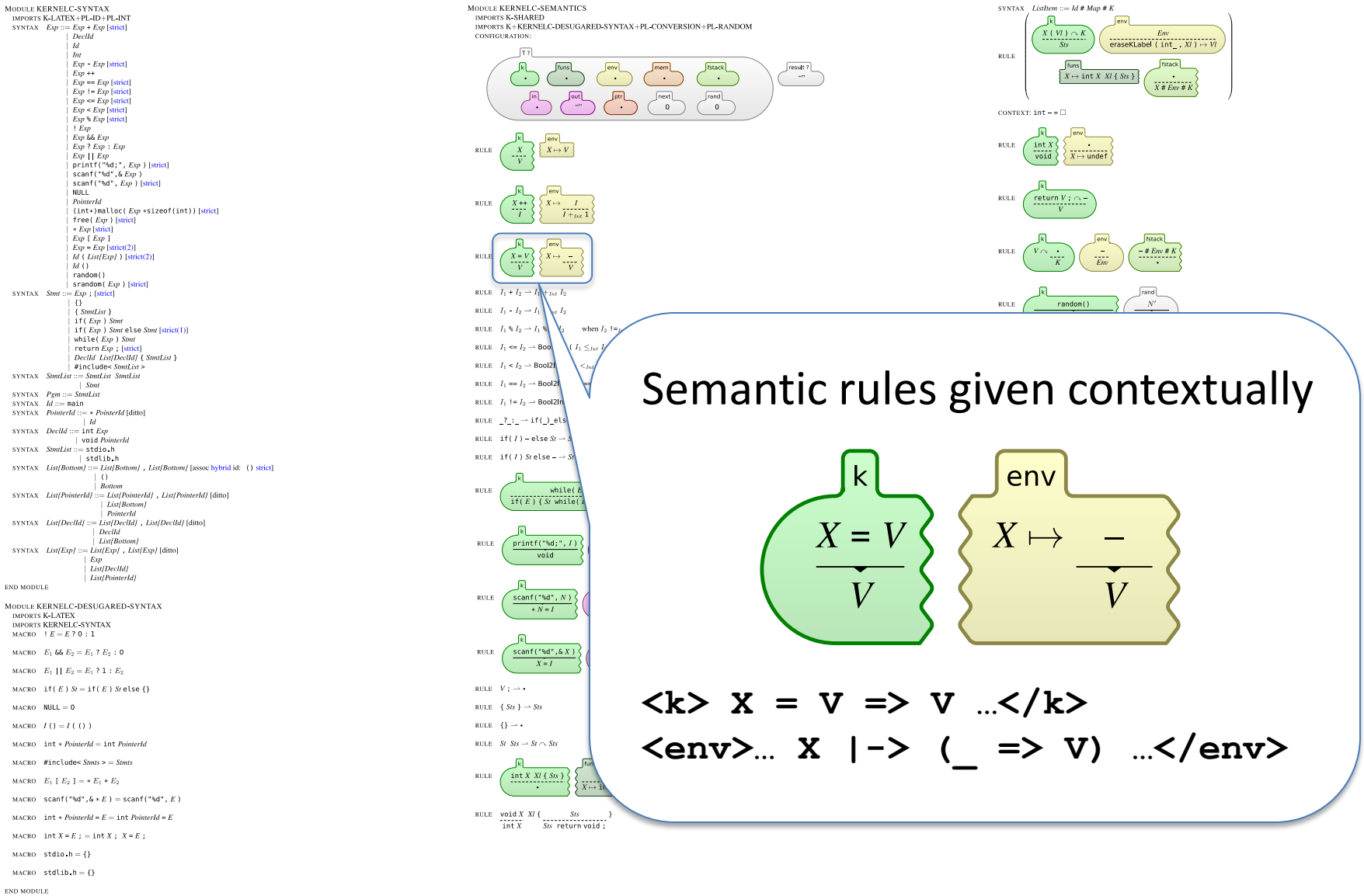
Complete K Definition of KernelC



Complete K Definition of KernelC



Complete K Definition of KernelC



K Semantics are Useful

- Executable, help language designers
- Make teaching PL concepts hands-on and fun
- Currently compiled into
  - Maude, for execution, debugging, model checking
  - Latex, for human inspection and understanding
- Plans to be compiled to
  - OCAML, for fast execution
  - COQ, for meta-property verification

Medium-Size K Definition

- See the semantics of SIMPLE on the “K and Matching Logic” page

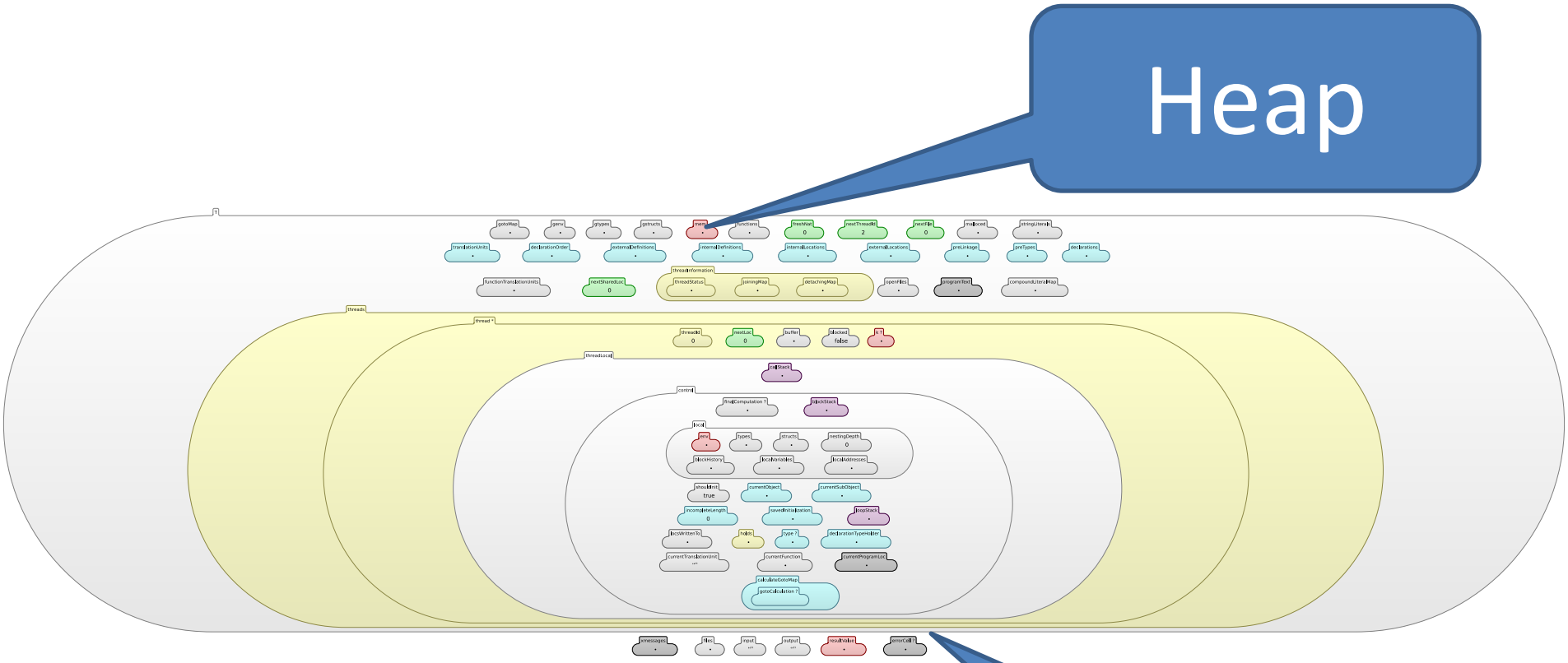
K Scales

Besides smaller and paradigmatic teaching languages, several larger languages were defined

- Scheme : by Pat Meredith
- Java 1.4 : by Feng Chen
- Verilog : by Pat Meredith and Mike Katelman
- C : by Chucky Ellison

etc.

The K Configuration of C



Statistics for the C definition

- Total number of rules: **~1200**

- Has been tested on thousands of C programs (several benchmarks, including the gcc torture test, code from the obfuscated C competition, etc.)
  - Passed **99.2%** so far!
  - GCC 4.1.2 passes 99%, ICC 99.4%, Clang 98.3 (no opt.)
- *The most complete formal C semantics*
- Took more than 18 months to define …
  - Wouldn't it be uneconomical to redefine it in each tool?

Matching Logic Verification

=

Rewriting (language semantics)

+

[FOL] (configuration reasoning)

+

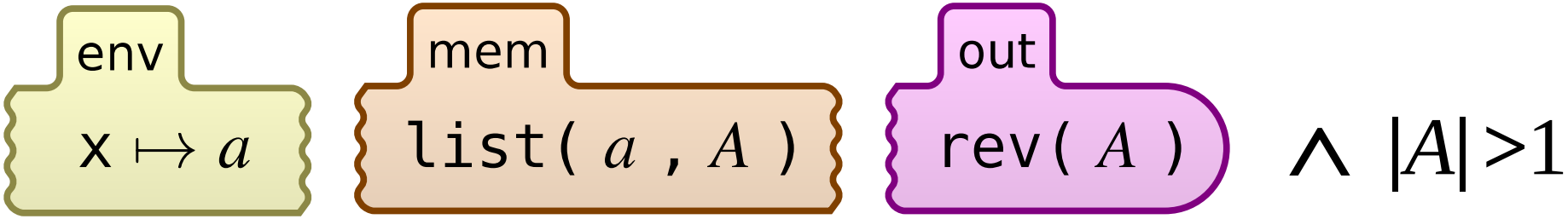
Proof Rules (behavior reasoning)

Matching Logic

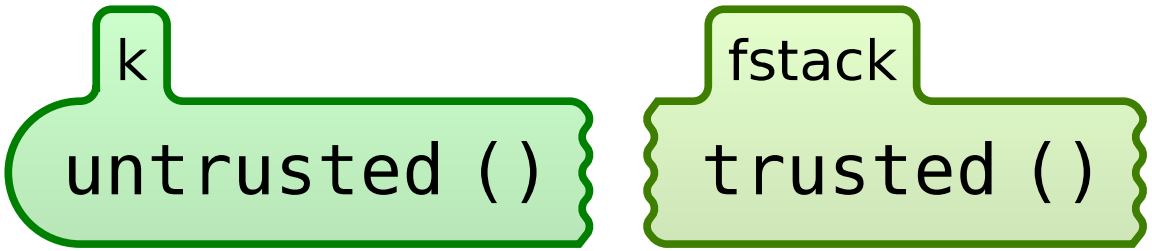
- A logic for reasoning about configurations
- Formulae
  - [FOL] over configurations, called patterns
  - Configurations are allowed to contain variables
- Models
  - Ground configurations
- Satisfaction
  - Matching for configurations, plus [FOL] for the rest

Examples of Patterns

- x points to sequence *A* with $|A|>1$, and the reversed sequence rev(*A*) has been output



- **`untrusted()`** can only be called from **`trusted()`**



More Formally: Configurations

- For concreteness, assume configurations having the following syntax:

$$\langle\langle \ldots \rangle_{k}\ \langle \ldots \rangle_{env}\ \langle \ldots \rangle_{heap}\ \langle \ldots \rangle_{in}\ \langle \ldots \rangle_{out}\ \ \ldots \rangle_{cfg}$$

(matching logic works with any configurations)

- Examples of concrete (ground) configurations:

$$\langle\langle \texttt{x=*y; y=x; } \ldots \rangle_{k}\ \langle \texttt{x} \mapsto 7,\ \texttt{y} \mapsto 3,\ \ldots \rangle_{env}\ \langle 3 \mapsto 5 \rangle_{heap}\ \ \ldots \rangle_{cfg}$$

$$\langle\langle \texttt{x} \mapsto 3 \rangle_{env}\ \langle 3 \mapsto 5,\ 2 \mapsto 7 \rangle_{heap}\ \langle 1, 2, 3, \ldots \rangle_{in}\ \langle \ldots, 7, 8, 9 \rangle_{out}\ \ \ldots \rangle_{cfg}$$

More Formally: Patterns

- Concrete configurations are already patterns, but very simple ones, ground
- Example of more complex pattern

$$\exists c : Cells,\ e : Env,\ p : Nat,\ \ i : Int,\ \sigma : Heap$$
$$\langle\langle \mathbf{x} \mapsto p,\ e \rangle_{\text{env}}\ \langle p \mapsto i,\ \sigma \rangle_{\text{heap}}\ c \rangle_{\text{cfg}}\ \wedge\ i > 0\ \wedge\ p \neq i$$

- Thus, patterns generalize both terms and [FOL]
- Models: concrete configurations + valuations
- Satisfaction: matching for patterns, [FOL] for rest

More Formally: Reasoning

- We can now prove (using FOL reasoning) properties about configurations, such as

$$
\begin{aligned}
\models \quad & \forall c : Cell,\ e : Env,\ p : Nat \\
& \langle\langle \mathbf{x} \mapsto p,\ e\rangle_{\mathsf{env}}\ \langle p \mapsto 9\rangle_{\mathsf{heap}}\ c\rangle_{\mathsf{cfg}}\ \wedge\ p > 10 \\
& \rightarrow \exists i : Int,\ \sigma : Heap \\
& \qquad \langle\langle \mathbf{x} \mapsto p,\ e\rangle_{\mathsf{env}}\ \langle p \mapsto i,\ \sigma\rangle_{\mathsf{heap}}\ c\rangle_{\mathsf{cfg}}\ \wedge\ i > 0\ \wedge\ p \neq i
\end{aligned}
$$

Matching Logic vs. Separation Logic

- Matching logic achieves separation through matching at the structural (term) level, not through special logical connectives (*)
- Matching logic realizes separation at all levels of the configuration, not only in the heap
  - the heap was only 1 out of the 75 cells in C's def.
- Matching logic can stay within FOL, while separation logic needs to extend FOL
  - Thus, we can use the existing SMT provers, etc.

Matching Logic as a Program Logic

- Hoare style - not recommended

$$\{\pi_{\text{pre}}\}\ \texttt{code}\ \{\pi_{\text{post}}\}$$

  – One has to redefine the PL semantics – impractical

- Rewriting (or K) style – recommended

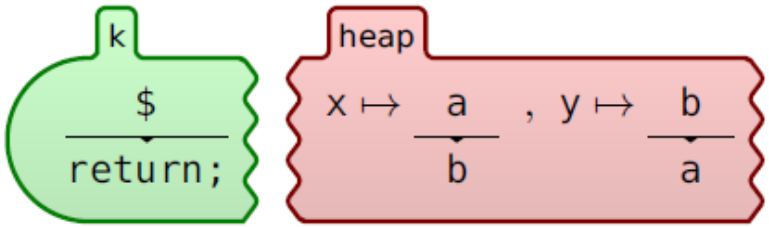
$$left[\texttt{code}] \rightarrow right$$

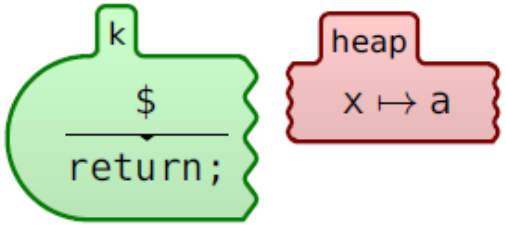
  – One can reuse existing K semantics – very good

Example – Swapping Values

```
void swap(int *x, int *y)
{
  int t;
  t=*x;
  *x=*y;
  *y=t;
}
```

$





if x = y

- What is the K semantics of the swap function?
- Let $ be its body

```
rule <k> $ => return; …</k>
     <heap>…
       x|->(a=>b),
       y|->(b=>a)
     …</heap>
```

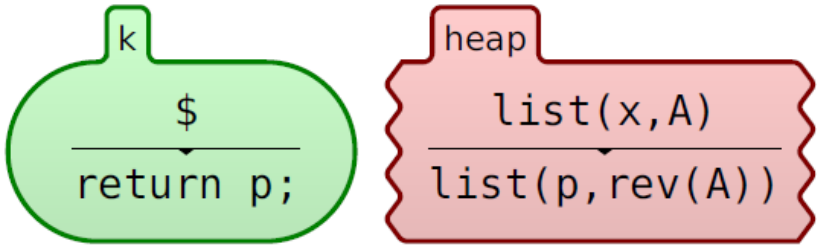
```
rule <k> $ => return; …</k>
     <heap>… x|-> a …</heap>
  if x = y
```

Example – Reversing a list

```
struct listNode* reverse(struct listNode *x)
{
  struct listNode *p;
  struct listNode *y;
  p = 0 ;
  while(x) {
    y = x->next;
    x->next = p;
    p = x;
    x = y;
  }
  return p;
}
```

$

- What is the K semantics of the reverse function?
- Let $ be its body




```
rule <k> $ => return p; </k>
     <heap>… list(x,A) => list(p,rev(A)) …</heap>
```

Partial Correctness

- We have two rewrite relations on configurations
  - $\rightarrow$ given by the language K semantics; safe
  - $\rightarrow$ given by specifications; unsafe, has to be proved
- Idea (simplified for deterministic languages):
  - Pick left $\rightarrow$ right. Show that always left $\rightarrow (\rightarrow \cup \rightarrow)^*$ right modulo matching logic reasoning (between rewrite steps)
- Theorem (soundness):
  - If left $\rightarrow$ right and "config *matches* left" such that config has a normal form for $\rightarrow$, then "nf(config) *matches* right"

More Formally: Matching Logic Rewriting

- Matching logic rewrite rules are rewrite rules over matching logic formulae: $\varphi \Longrightarrow \varphi'$
- Since patterns generalize terms, matching logic rewriting captures term rewriting
- Moreover, deals naturally with side conditions: rewrite rules of the form

$$l \Longrightarrow r \text{ if } b$$

are captured as matching logic rules of the form

$$l \wedge b \Longrightarrow r$$

More Formally: Proof System I

- Rules of operational nature

Reflexivity

$$\frac{\cdot}{\mathcal{A} \vdash \varphi \Rightarrow \varphi}$$

Axiom

$$\frac{\varphi \Rightarrow \varphi' \in \mathcal{A}}{\mathcal{A} \vdash \varphi \Rightarrow \varphi'}$$

More Formally: Proof System II

Substitution

$$\frac{\mathcal{A} \vdash \varphi \Rightarrow \varphi' \qquad \theta : Var \to \mathcal{T}_\Sigma(Var)}{\mathcal{A} \vdash \theta(\varphi) \Rightarrow \theta(\varphi')}$$

Transitivity

$$\frac{\mathcal{A} \vdash \varphi_1 \Rightarrow \varphi_2 \qquad \mathcal{A} \vdash \varphi_2 \Rightarrow \varphi_3}{\mathcal{A} \vdash \varphi_1 \Rightarrow \varphi_3}$$

More Formally: Proof System III

- Rules of deductive nature

Case analysis

$$\frac{\mathcal{A} \vdash \varphi_1 \Rightarrow \varphi \qquad \mathcal{A} \vdash \varphi_2 \Rightarrow \varphi}{\mathcal{A} \vdash \varphi_1 \vee \varphi_2 \Rightarrow \varphi}$$

Logic framing

$$\frac{\mathcal{A} \vdash \varphi \Rightarrow \varphi' \qquad \psi \text{ is a FOL}_{=} \text{ formula}}{\mathcal{A} \vdash \varphi \wedge \psi \Rightarrow \varphi' \wedge \psi}$$

More Formally: Proof System IV

Consequence

$$\frac{\models \varphi_1 \rightarrow \varphi_1' \qquad \mathcal{A} \vdash \varphi_1' \Rightarrow \varphi_2' \qquad \models \varphi_2' \rightarrow \varphi_2}{\mathcal{A} \vdash \varphi_1 \Rightarrow \varphi_2}$$

Abstraction

$$\frac{\mathcal{A} \vdash \varphi \Rightarrow \varphi' \qquad X \cap FreeVars(\varphi') = \emptyset}{\mathcal{A} \vdash \exists X\, \varphi \Rightarrow \varphi'}$$

More Formally: Proof System V

- Main proof rule of matching logic rewriting

Circularity

$$\frac{\mathcal{A} \vdash \varphi \Rightarrow^{+} \varphi'' \qquad \mathcal{A} \cup \{\varphi \Rightarrow \varphi'\} \vdash \varphi'' \Rightarrow \varphi'}{\mathcal{A} \vdash \varphi \Rightarrow \varphi'}$$

Fact

- Matching logic generalizes both operational semantics and axiomatic semantics
  - Operational semantics by means of capturing term rewriting as discussed above
  - Axiomatic semantics by noticing that Hoare triples are particular pattern rewrites:

$$\mathrm{HL2ML}(\{\psi\}\ \mathtt{s}\ \{\psi'\}) \quad =$$

$$\langle \mathtt{s},\ \sigma_Z \rangle \wedge \sigma_Z(\psi) \Rightarrow \exists Z(\langle \mathtt{skip},\ \sigma_Z \rangle \wedge \sigma_Z(\psi'))$$

Theorem

- Any operational behavior can also be derived using matching logic reasoning
- For any Hoare triple $\{\psi\}\ \mathtt{s}\ \{\psi'\}$ derived with axiomatic semantics, the corresponding matching logic rule $\mathrm{HL2ML}(\{\psi\}\ \mathtt{s}\ \{\psi'\})$ can be derived with the matching logic proof system
  - Proof is constructive, not existential
- Partial correctness
  - Holds for ALL languages

MatchC

- A Matching Logic Verifier for (a fragment of) C
- Uses the K semantics of the C fragment *unchanged*
- Has verified a series of challenging programs
  - Undefiness, typical Hoare-like programs, heap programs (lists, trees, stacks, queues, graphs), sortings, AVL trees, Schorr-Waite graph marking

Conclusions

- K (semantics) and Matching Logic (verification)
- Formal semantics is useful and practical!
- One can use an executable semantics of a language *as is* also for program verification
  - As opposed to redefining it as a Hoare logic
- Giving a formal semantics is not necessarily painful, it can be fun if one uses the right tools